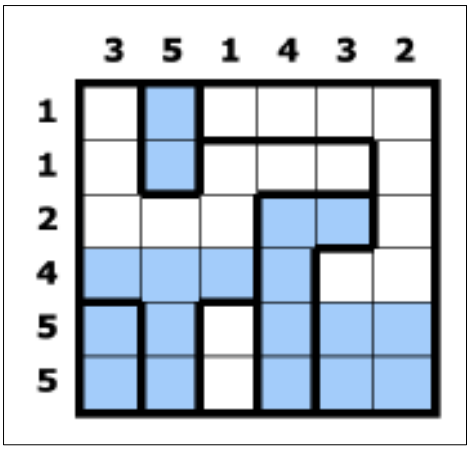
Representing the Game as a Matrix:

In order to, represent the game as a Matrix we need to assign different values to the different cells. Every cell in the game is a Variable of CSP and their domains are always two valued as empty or filled. I used 3 main values for in the matrix, as following:

- 0 indicates the cell is a Variable and each one of the original cells in the game becomes this variable as we will try to determine their values (empty or filled)
- X indicates the cell walls as we need to represent them somehow, I directly included them in the matrix but this leads another problem, some of the non-walled cells expand due to this additional rows and columns created by the walls, to handle that I defined the third matrix element.
- F indicates the cell is a free cell, that occurred due to the expansion caused by X's, and the meaning of it is if there is a F between two 0's (real cells), they are directly connected without any restrictions.

So, the overall matrix became the following:



F	F	3	F	5	F	1	F	4	F	3	F	2
F												
1		0	X	0	X	0	F	0	F	0	F	0
F		F	X	F	X	X	X	X	X	X	X	F
1		0	X	0	X	0	F	0	F	0	X	0
F		F	X	X	X	F	X	X	X	X	X	F
2		0	F	0	F	0	X	0	F	0	X	0
F		F	F	F	F	F	X	F	X	X	X	F
4		0	F	0	F	0	X	0	X	0	F	0
F		X	X	F	X	X	X	F	X	F	F	F
5		0	X	0	X	0	X	0	X	0	F	0
5		0	X	0	X	0	X	0	X	0	F	0

Figure 1: 6x6 puzzle given in the document and representation of it as matrix by defined rules above (separated and coloured for better visualization.).

However, for better handling in the Python and Google OR-Tools, F and X's are changed to numbers as suggested by OR-Tools to work with integers. X's are changed to 9 and F's are changed to 8. And the matrix given to the program is as following:

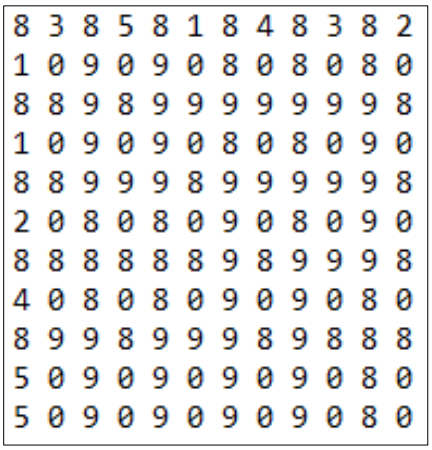


Figure 2: Numerated version of the matrix in figure 1, that is given to the program.

Defining Constraints:

The constraints of the game are basic. There are mainly three constraint categories with second and third being together due to gravity mechanism. Separating the second and the third gave a better opportunity to handle the problem.

- First one is, each row and column should have as many filled cells as the indicated value in the beginning of that row or column. If that row or column starts with F it means that row/column is a virtual row created due to expansion caused by cell walls, and that row is row/column is free and can be skipped during reading.
- Second constraint is, if a cell is filled, every connected (either directly or by a free cell F), all connected cells should be also filled. And same applies for cells being empty. Basically, each horizontally connected cell should have equal value.
- Lastly, if a cell is connected to a cell under it (either directly or a by free cell F) and if the cell at the bottom is empty the cell at the top should be empty too. And if the cell at the bottom is filled, the cell at the top can be either empty or filled.

A* Search and CSP Solver:

In this problem using the, CSP Solver over A* is more appropriate and there are several reasons for this.

Firstly, we do not require a sequence of filling blocks or even need to consider states that has less or more filled cells than required number; while this might be still the case in OR-Tools we used, it still does not change that this process is unnecessary and don't need to be computed adding a point to why standard search algorithms that blindly or biasedly expanding a state after state should not be used.

On top of this problem, having no constraint checks and having a goal test instead makes the search much more complex. Even if we might have lots of constraints that would take lots of time to condition check, being able to detect violations of constraints and stopping the search towards that direction is a huge advantage, becoming even more important in bigger spaces; for example if A* search does a constraint violation in the first assignment it did, it will still expand the search above that state and will not stop until all matrix variables are assigned to a value and checking whether a goal is reached or not. In CSP solvers we have this advantage of foreseeing these violations and avoid wasting time on expanding over those problematic states.

An additional advantage of CSP solvers (over being able to stop search in such bad directions earlier) would arise with the increasing number of constraints. Even the increasing number of constraints will mean an increase in the condition checks, if's and else's leading to a longer compiling time of a single step in the search loop, more constraints would mean less legal values that can be assigned to a variable. And therefore, search space would shrink in size with the increasing number of constraints, leading to a faster and less space taking search.

Reading the Output of the Program:

As the program gives an unorganized output as a solution and even if it was organized it is hard to visualize it similar to the matrix representation of the game. More understandable and a clearer way of reading the result is as following.

```
Status = OPTIMAL
[0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1]
```

Figure 3: The result lines of the output of the program to the 6x6 puzzle in the figure 1. (Each 6 element is separated for better visualisation of the solution)

The result of the program has the following format as shown in figure 3 consists of two parts.

1. The text in the first line indicates status of the search of the space:
   - "OPTIMAL" means all solutions are found
   - "FEASIBLE" means some solutions (or none) are found at the current state of the search but there exists at least one solution to be found in the space.
   - "INFEASIBLE" means no solution is found at the current state of the search and there is no solution to be found in the space.

   Since the aquariums have exactly one solution to the problem given, at the end of the search status is always "OPTIMAL". Assuming the puzzle is a valid one, having no ambiguity with several solutions and it does have a solution as supposed.

2. The array given indicates the values given to the variables by CSP solver. To read it better one can separate the array to n element pieces for a given nxn puzzle. In this case the puzzle is 6x6 so the elements of the array is stripped into 6 element pieces in the figure 3 by red lines. Then starting from left each strip corresponds to a row of the puzzle. Therefore, the solution given by the program in puzzle format is as following:

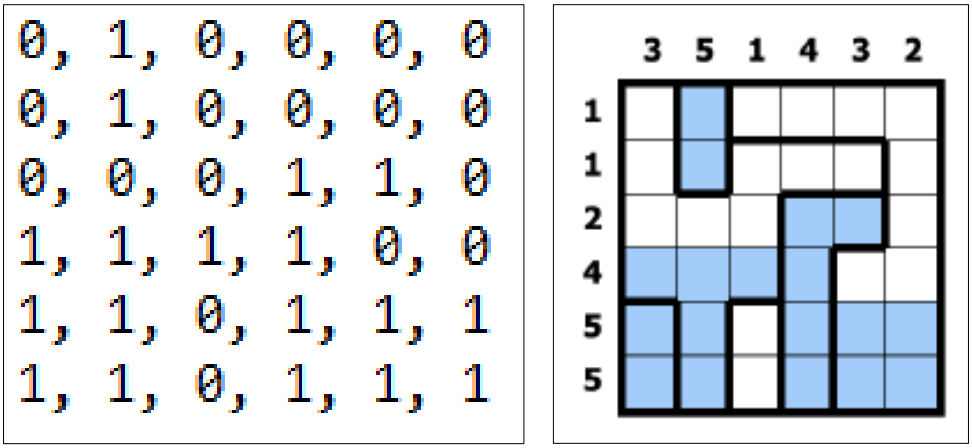
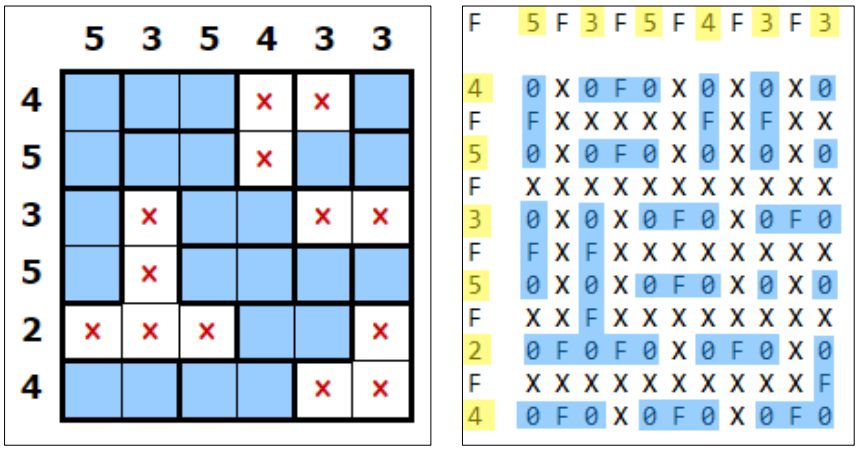


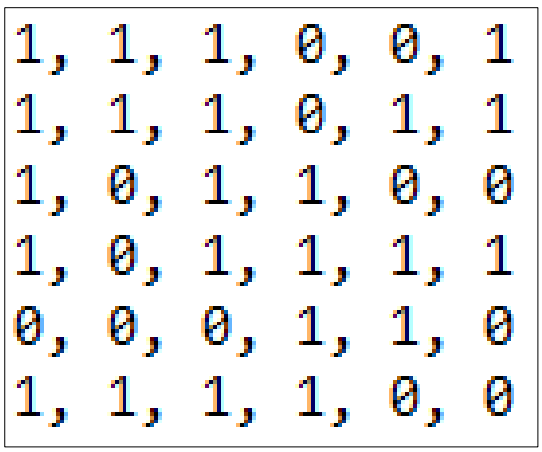
Figure 4: Solution of the program to the 6x6 puzzle in the figure 1 and the puzzle itself.

As it can be seen better with this 6x6 format, instead of 1x36, each number refers to a cell in the puzzle and 1's indicate the filled cells while 0's indicate empty cells. And it can be seen that the solution given by the program matches with the actual solution, hence it works fine.

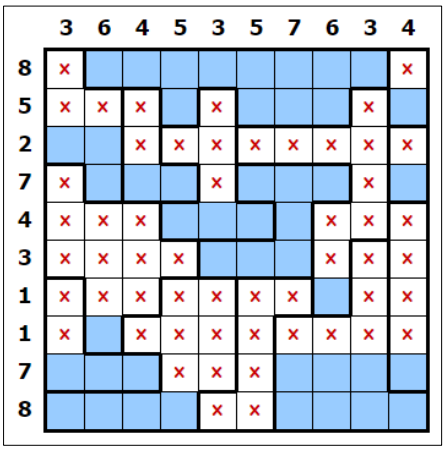
Additional puzzles used:

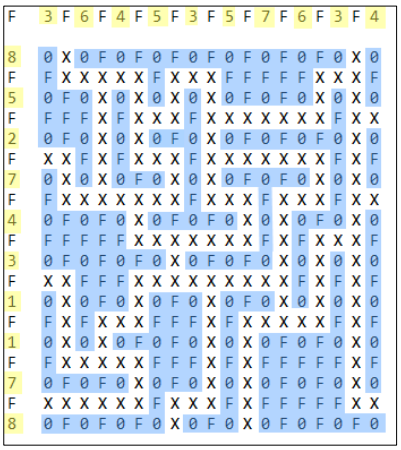
6x6 hard





10x10 easy





```
0, 1, 1, 1, 1, 1, 1, 1, 1, 0
0, 0, 0, 1, 0, 1, 1, 1, 0, 1
1, 1, 0, 0, 0, 0, 0, 0, 0, 0
0, 1, 1, 1, 0, 1, 1, 1, 0, 1
0, 0, 0, 1, 1, 1, 1, 0, 0, 0
0, 0, 0, 0, 1, 1, 1, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 1, 0, 0
0, 1, 0, 0, 0, 0, 0, 0, 0, 0
1, 1, 1, 0, 0, 0, 1, 1, 1, 1
1, 1, 1, 1, 0, 0, 1, 1, 1, 1
```